"A Tale of Two Provers" :

Comparing Program Verification Techniques in Dependent Haskell and F*

Xinyue Zhang (xzhang@brynmawr.edu)

Advised by Richard A. Eisenberg

February, 2018

1 Introduction

We as programmers write many programs every day, but how do we make sure that these programs indeed perform as intended, and how can we guarantee that these programs cannot be used to perform unintended tasks ?

Program verification is a technique, which uses formal mathematical methods to prove the correctness of a software program or system, with respect to a formally defined or inferred specification. It can detect most system vulnerabilities and program bugs at compile time. There are two mainstreamed approaches to verify programs : those based on Type Theory (TT), as in Coq[3], Agda [8], and Idris[2] [1], and others based on Satisfiability Modulo Theory (SMT), such as Liquid Haskell [10], Dafny[6], and F*[9].

TT-based Dependent Haskell and SMT-based dependent-and-refinement-typed F* each represents a unique and effective program verification technique. They are similar in that they both aim towards a verification-based yet general-purposed programming language, but they are noticeably different in their unique approaches to categorize programming effects. Working in both languages for a semester, I find promising properties that I believe might lead to potential collaborations. However, to my knowledge, and with consultation with Eisenberg and Swamy, the two communities generally conduct research separately and have rarely explored each other's approaches in-depth.

In my undergraduate thesis, I aim to introduce the two verification-oriented programming languages, namely Dependent Haskell and F*, present their unique type system designs with concrete examples, and give a detailed comparison from a combination of theoretical and practical standpoints. I expect my research to setup a solid foundation for potential collaborations across the two communities, and to possibly propose promising future research directions in program verifications for the general Programming Languages community.

1. Specifically, all three languages are based on Martin-Löf Type Theory

2 Background and Related Work

Haskell is a general-purposed, strongly statically typed, and purely functional programming language. It features a type system with type inference and lazy evaluation. For a long time, Haskell researchers want to add dependent types into Haskell, as it introduces much more precise expressions of program specifications. Although Haskell hasn't yet supported dependent types in full, several extensions on its current type system have make exceptional progresses. In 2012, Eisenberg and Weirich introduce the `singletons` library that essentially achieves dependently-typed programming in Haskell[5]. With the common goal of introducing full-fledged dependently-typed Haskell, Gundry and Eisenberg are both interested in GADTs and type families. Gundry contributes a rebuilt algorithm for first-order unification and Hindley-Milner type inference, which is applied in the constrained solver underlying the elaboration algorithm behind Gundry's implementation. Eisenberg, extending Gundry's work, goes one step further to blur the distinction for type and kind levels, unifying expressions and types in Dependent Haskell[4]. Besides type families that encode type-level programming through intuitive functions and is widely adopted in Haskell nowadays, functional dependencies is another, indeed the earliest introduced approach to enable rich type-level programming through relations. In 2017, Morris and Eisenberg acknowledged in Constrained Type Families that the current Haskell type system inevitably accepts some apparently erroneous definitions, as type families equivocally assumes totality. They propose the design of constrained type families to relax the totality assumption by characterizing domains of partial type family definitions using type class predicates[7].

F* is an ML-like functional programming language aimed at program verification and is generally compiled into OCaml, F#, or C for efficiency[9]. At POPL 2016, Swamy et al. introduced the current, completely redesigned version of F*, a language that works as a powerful and interactive proof assistant, a general-purpose, verification-oriented and effectful programming language, and a SMT-based semi-automated program verification tool. F* is a dependently typed, higher-order and call-by-value language with a lattice of primitive effects. These monadic effects can be specified by the programmer and is used by F* to discharge verifications using both SMT solver and manual proofs[9]. In addition, it uses a combination of dependent types and refinement types for program verification and it supports termination checking based on a metric of well-founded decreasing order to ensure program consistency. Dependently typed F* is further enhanced by the introduction of Dijkstra Monads to perform program verification on effectful code beyond the primitive effects by Ahman et el. at POPL 2017[1].

3 Approach and Uniqueness

To gain a better understanding of the language design decisions and verification techniques, I first carefully study 2-3 literals for each language. In general, my assessment of Dependent Haskell and F* is based on implementing and proving a wide spectrum of algorithms in both languages. These programs are written so that I could have a much more in-depth experience with both languages and explore their features more thoroughly. To begin with, in both Dependent Haskell and F*, I implement the data type of a length-indexed vector and verify various functions that support types of finite length vectors from Haskell's List module, including length, head, tail, init, last, snoc, reverse, and, or, null etc. I then focus on the verification for various sorting algorithms, such as mergesort and quicksort, and examine the capability of proving divergent functions, as in the proof of peano division with zero divisor. I structure the comparison mainly focusing on three aspects : language design measured from syntactic verbosity and type system expressiveness, and library/documentation supports ; verification techniques determined through the adaptability and effectiveness of proofs ; and finally termination checking evaluated using quantified success rates as metric. As far as I know, I am the first researcher to directly compare program verification techniques between Dependent Haskell and F*.

4 Current Results

At the 45th ACM POPL Student Research Competition, Xu and Zhang presented their collaborated work on the comparison among three programing verification techniques in Dependent Haskell, Liquid Haskell and F*. They conclude that among the three languages, Dependent Haskell has the most verbose syntax, but at the same time is comparably more expressive. Liquid Haskell and F*, both provide a more friendly user interface, but Liquid Haskell has some consistency issues and F* has difficulties in proving divergent functions. Liquid Haskell, instead of taking the type theory approach as Dependent Haskell, introduces refinement reflection and in turn supports SMT-solver automation. F*, combining the benefits of both type theory based dependently typed lemma proofs and SMT automated verifications, supports both explicit and implicit programming. Finally, contrary to Liquid Haskell and F* and indeed most dependently typed programming languages, Dependent Haskell doesn't require totality checking at compile time and instead verifies proof termination at run-time. As a partial language, it may have proofs that diverges or errors but it guarantees correctness of fully evaluated proofs. As a result, they advocate more collaborations between Dependent Haskell and Liquid Haskell, using F* as a reference.

Références

[1] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17. ACM, 2017.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language : Design and implementation. *J. Funct. Prog.*, 23, 2013.

[3] Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[4] Richard A. Eisenberg. *Dependent Types in Haskell : Theory and Practice.* PhD thesis, University of Pennsylvania, 2016.

[5] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[6] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[7] J Garrett Morris and Richard A Eisenberg. Constrained type families. volume 1, page 42. ACM, 2017.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[9] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[10] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.